Flight Software Branch
Ada Coding Standard

Stephen Leake

30 January 2004

1 Introduction

This document presents the coding standards used by Goddard Flight Software branch projects for the Ada language. There are two primary goals for these coding standards:

- Provide a common style for Ada code, so it can be comfortably read by all programmers on the project.
- Make the Ada language easier to use, by giving guidelines on using complex features.

This standard sets the policy for the appropriate use of Ada language constructs, identifier naming conventions, and general source code layout. It also suggests coding patterns for use with Ada.

The standards described in this document are either mandatory or discretionary. Mandatory standards are requirements that must be followed. These are indicated by the word "shall". Discretionary standards are guidelines that allow some judgment or personal choice by the programmer. These standards are indicated by the use of the word "should". A programmer should have a good reason when they choose to disregard a discretionary standard. Where possible, this document gives examples of acceptable deviations. A mandatory standard can be waived by the development lead for specific cases, where appropriate.

In this document, paragraphs labeled by a number in parentheses (n) are standards; other paragraphs are explanatory text.

The primary mode of reading code is assumed to be on a computer screen, with an editor that provides syntax colorization and code navigation. There will be a few times, primarily at code reviews, when code will be printed out. The coding standards reflect this bias towards on-screen viewing.

It is easier to combine code into a system, and to maintain the code, if all of the code conforms to a particular style. Since Ada is new to many programmers, and the general Ada community shares a fairly standard style, we require a particular style. See Figure 1 for an example of the style.

2 Language

1. The code shall only use Ada language constructs as defined by ISO/IEC 8652:1995: Information Technology – Programming Languages – Ada. An exception is allowed if a preprocessor is needed for portability; then the constructs defined by the GNAT preprocessor `gnatprep` are allowed.

2. Non-standard pragmas shall require a waiver and full documentation.
3. The non-standard GNAT pragma "Unreferenced" is granted a waiver.
4. All source files shall consist of printable ASCII characters, with no tabs or form feeds, and no trailing spaces on lines, maximum of 120 characters per line.

3 File and Package Naming Conventions

3.1 File names

1. Each source file shall contain one compilation unit.
2. The filename shall be the full Ada unit name, all lowercase, with "." replaced by "-". This is the default naming convention used by the GNU Ada compiler (GNAT). An exception is allowed when different bodies are required for different targets; in that case the filename shall be the standard file name, with a suffix indicating the target.
3. Specification files shall have an extension of `.ads`. Body files shall have an extension of `.adb`.
4. Files that contain preprocessor constructs shall have a second extension of `.gp` (for gnatprep). Thus the file `foo.ads.gp` is processed by gnatprep to produce the file `foo.ads`.

3.2 Package names

1. If unit test drivers are a child procedure of the unit package, they shall have the name `Test`. For example, the unit test driver for package `Foo` is `Foo.Test`. This gives the unit test access to the private part of the package.
2. Generic packages shall have a name that starts with `Gen_` or `Generic_`.
3. Unit tests for generic packages may either be children of a nominal instantiation of the package, or a main procedure that instantiates the package. If the latter, the name shall drop the generic prefix, and add a `_Test` suffix. So a test for `SAL.Filters.Gen_First_Order` would be named `SAL.Filters.First_Order_Test`.

4 File layout

1. The file shall be introduced by a block comment that contains a file prologue (see Figure 1). The file prologue consists of:

   Abstract A short statement that describes the purpose of the file. In a package body, the abstract should just refer to the specification.

   References List all applicable documentation references. Identify each reference to a level of detail that allows the information to be found but not to a level that will likely change. Provide version numbers when applicable.

   Design Notes List design decisions that apply to the general design of the code. Give a brief justification for each decision; for example, list the alternatives and say why they were rejected. Longer justifications may be given in a separate design document.

   Copyright Copyright or disclaimer notice.

2. Context clauses shall list one package per line. If a "use" clause is present, it should be on the same line as the corresponding "with".

3. Declarations should be grouped logically. A group label comment should be used to separate logical groups:

   ```
   ----------
   -- Orientation operations
   ```

4. When there are more than five subprograms in a group, they should be listed alphabetically within the group.

5. Operations of tagged types should be grouped into "class-wide" and "dispatching" groups.

6. Operations of derived types should be grouped into "overriding" and "new" groups.

Figure 1: Example package specification file

```
-- Abstract:
--
-- General utilities for star tracker models.
--
-- References:
--
-- [1] Star Tracker Component
--
-- Design Notes:
--
-- Catalog_Type is abstract tagged to provide a common interface to different
-- catalog implementations.
--
-- Disclaimer
--
-- <standard NASA disclaimer>
--

with Ada.Numerics.Float_Random;
with Math_3_DOF;
with Math_Scalar;
package HDS.Star_Tracker is

   ----------
   -- Orientation operations

   function Aberration
     (GCI_q_ST        : in Math_3_DOF.Unit_Quaternion_Type;
      ST_Boresight    : in Math_3_DOF.Unit_Vector_Type     := Math_3_DOF.Z_Unit;
      GCI_Sun_v_Earth : in Math_3_DOF.Cart_Vector_Type;
      GCI_Earth_v_Sc  : in Math_3_DOF.Cart_Vector_Type)
     return Math_3_DOF.Unit_Quaternion_Type;
   -- Return relativistic velocity aberration quaternion ST_q_Ab.
```

Figure 2: Example package specification file continued

```
   ---------------
   -- Star vector operations

   type Star_ID_Type is new Integer;

   Unknown : constant Star_ID_Type := -1;

   type Star_Vector_Type is record
      ID        : Star_ID_Type                 := Unknown;
      Vector    : Math_3_DOF.Unit_Vector_Type := Math_3_DOF.X_Unit;
      Magnitude : Math_Scalar.Float_Type      := 0.0;
   end record;

   Unknown_Star : constant Star_Vector_Type :=
     (ID        => Unknown,
      Vector    => Math_3_DOF.X_Unit,
      Magnitude => 0.0);

end HDS.Star_Tracker;
```

5 Style enforcing tools

Style rules are easier to follow when tools are available to enforce them, or automate using them. We define the required style in part by the tools given here. The style requirements given in the rest of this document should conform to this definition; however, when there is a conflict, the tool is correct.

1. Code shall be indented as defined by the GNU Emacs Ada mode, as distributed with Emacs (version 21.2 or higher), or as distributed by ACT, with the following settings (other settings have their default values):
   - (setq ada-when-indent 0)
   - (setq ada-label-indent 0)
2. Groups of statements, record declarations, and named parameter associations shall be aligned by the standard Emacs package 'align.el.
3. Code shall conform to the GNAT (version 3.16a or higher) style check compiler option "-gnaty3abefhiklM120nprt". See the GNAT Users Guide for more information.

6 Comments

Comments are important in developing readable and maintainable code. Programmers should include comments whenever it is difficult to understand the code without the comments.

On the other hand, comments should not simply say the same thing the code does, as this only serves to obscure required comments. Describe why something is being done first, and only describe what is being done if it isn't obvious by the code itself.

Comments can be classified by size:

- Long or block comments are those that can not fit on a single 80 character line.
- Short comments are those that can fit on a single 80 character line (note that this is shorter than the code line length limit).
- Same-line comments are small enough to include on the same line as the code that the comment supports.

1. Functional blocks of code should have a long comment before the actual code instead of placing a comment on each line. This comment should describe the basic purpose of the code. The comment should consist of complete English sentences.
2. A variables units of measurement should be stated in a comment if they are not the SI standard units, as defined at http://www.bipm.fr/enus/.
3. Change log comments shall not be in the source code; use a configuration management tool instead.
4. Old versions of the code shall not be maintained in the comments; use a configuration management tool instead.
5. A special comment format shall be used to document compiler workarounds. This allows finding and fixing the workarounds when a new compiler version is released.

   ```
   --  WORKAROUND: <compiler version> <description>
   ```

6. A special comment format shall be used to document an incomplete implementation:

   ```
   --  FIXME: <description>
   ```

7 Identifier Naming conventions

The proper naming of packages, types, subprograms and variables can increase the readability and understanding of the software. Poorly named functions and variables can add a great deal of misunderstanding and confusion. Careful consideration should be given to how others will perceive the name, and what they will think it means. An object‘s name should provide insight into its use and purpose.

1. Identifiers should consist of more than one character. Valid exceptions are for variables used in local indexing operations (e.g., i, j, k) and very mathematical code where the identifier matches the standard symbol used in an algorithm or equation.
2. Ada reserved words shall be all lowercase, except when used as attributes.
3. Acronyms shall be all upper case. Each project shall maintain a list of approved acronyms.
4. All other identifiers shall be mixed case, with underscores separating words. Acronyms that are part of a larger identifier shall be uppercase. This style is often called `Mixed_Case_With_Underscores`.

5. The names of all types not defined by the language shall end with a suffix of `_Type`. This allows using the same general name for the type and an object: `List :  List_Type;`

6. Capitalization of identifiers shall be consistent among all occurrences of the identifier. The GNAT style check compiler option checks this.

7. The optional name at the end of a named construct (ie. procedures, functions) shall always be present.

8 Indentation

1. All code shall be indented three spaces for each indentation level.

2. Statements at the same logical nesting level shall be at the same indentation level.

3. The indentation of a long or short comment shall be the same as the code it describes.

4. There should be only one statement per line.

5. Blank lines should be used between blocks of code that are functionally distinct.

6. In expressions, operators should be surrounded by blanks, except for prefix operators. This is enforced by the GNAT style checks.

7. Long lines should be broken after operators or commas.

8. Continuation lines shall be indented one level.

8.1 Subprogram Specifications

1. A procedure specification shall have the one of the following formats:

   ```
   procedure Name (Arg_1 : <mode> <type>; Arg_2 : <mode> <type>);

   procedure Name
     (Arg_1 : <mode> <type [:= <default>];
      Arg_2 : <mode> <type [:= <default>]);
   ```

2. A function specification shall have the one of the following formats:

```
function Name (Arg_1 : <mode> <type>; Arg_2 : <mode> <type>) return result_<type>;

function Name
  (Arg_1 : <mode> <type> [:= <default>];
   Arg_2 : <mode> <type> [:= <default>])
  return <result_type>;
```

However, the mode is optional in functions, since it must always be "in".

3. All multi-line parameter lists shall have the format given by the Emacs Ada mode align function.
4. Each subprogram declaration shall be followed by a block comment giving a description of the subprogram, including its purpose and how each argument is used. For simple subprograms with clear argument names, the descriptive comment can be empty. For a very complex subprogram, the description should just reference separate documentation.

8.2 Statements

8.2.1 Variable declarations

1. Each variable declaration shall declare only one variable. For example:

```
 One_List     : List_Type;
 Another_List : List_Type;
```

not:

```
 One_List, Another_List : List_Type; -- wrong!
```

8.2.2 if Statements

1. An if statement shall have one of the following formats:

```
if condition then
   statement;
[elsif condition then
   statement;]
else
   statement;
end if;
```

```
if long-condition
   continued-condition
then
   statement;
[elsif
   long-condition
then
   statement;]
else
   statement;
end if;
```

The else clause is optional. These formats are checked by the GNAT style check.

8.2.3 case Statement

1. A case statement shall have the following format:

```
case expression is
when constant_expression_1 =>
    statements;
when constant_expression_2 =>
    statements;
end case;
```

2. A `when others` clause shall only be present if the case statement is not exhaustive. Most case statements should be exhaustive, so the compiler will warn you when a value is added to the case expression type.

8.2.4 Miscellaneous

1. Named association shall be used for subprogram parameters and aggregates whenever the parameter order is not fully determined by the types. For example, if a subprogram has two parameters of type Integer, named association must be used to distinguish them.

2. Named association shall be used for all generic instantiations.

3. A use clause may be given for a package only in a package or subprogram body, not in a package specification. The scope of the use clause should be limited as much as reasonable.

   Use clauses make for more compact code, with the tradeoff of making it more difficult to understand where identifiers are declared. In package specifications, it is more important to understand the relationship with other packages, so use clauses are not used. In bodies, it is more important to allow for compact code, which makes it easier to understand the control flow.